**UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA**

**FACULTATEA DE MATEMATICĂ ŞI**

**INFORMATICĂ SPECIALIZAREA INFORMATICĂ**

**GERMANĂ**

# LUCRARE DE LICENŢĂ

## Plannier
## O aplicație Flutter pentru planificarea și stocarea momentelor frumoase

**Conducător ştiinţific**

**Lect. Univ. Dr. Cristea Diana**

*Absolvent*

Blidar Amalia-Bianca

**2023**

**BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA**
**FACULTY OF MATHEMATICS AND COMPUTER SCIENCE**
**SPECIALIZATION COMPUTER SCIENCE IN GERMAN**

# DIPLOMA THESIS

## Plannier

## A Flutter application for planning and storing beautiful moments

**Supervisor**
**Lecturer Diana Cristea, PhD**

*Author*

Blidar Amalia-Bianca

**2023**

**BABEŞ-BOLYAI UNIVERSITÄT CLUJ-NAPOCA**
**FAKULTÄT FÜR MATHEMATIK UND INFORMATIK**
**INFORMATIK IN DEUTSCHER SPRACHE**

# BACHELORARBEIT

## Plannier
## Eine Flutter-Anwendung zum Planen und Speichern schöner Momente Ihrer Veranstaltung

**Betreuer**

**Lektor Dr. Cristea Diana**

*Eingereicht von*

Blidar Amalia-Bianca

**2023**

# Abstract

This bachelor thesis explores plugin development in Flutter, an open-source UI software development kit by Google designed to create consistent cross-platform applications. The purpose of the thesis, as mentioned in the first chapter, is to understand the process of plugin development within the Flutter framework, which enables developers to use native platform features and improve their applications. The study also focuses on the YUV color space and manual image-to-video conversion, exploring the technical details and gaining a deeper understanding of the conversion process. The conversion and development of the plugin are presented in detail in the second chapter. The third chapter introduces all the technologies used for development, while enhancing their advantages and disadvantages. The context for plugin development is the Plannier event management application, designed to help users manage tasks, send or receive event invitations, and preserve event memories by converting images to videos. The application's design and architecture, as well as all the functionalities which it contains are presented in the fourth chapter. The motivation for developing a plugin in Plannier results from the need to efficiently access modern native APIs for image-to-video conversions, because existing solutions rely on outdated APIs. The FlutterMediaWriter Plugin facilitates the conversion process by transforming JPG-format photos into YUV_420_888, ensuring video compatibility and efficient compression. The fifth chapter is meant to provide an overall look at the goals that were achieved and to present future development possibilities. Finally, the sixth chapter of this bachelor thesis compiles all the references that have been utilized throughout its creation, providing a comprehensive list of the sources that have informed and supported the research conducted. Overall, this thesis provides insights into Flutter plugin development and its application in the event management domain.

# Contents

# 1. Introduction

## 1.1. Purpose and Motivation

Planning and managing events can be a challenging task, often accompanied by stress and the need for effective organization (Doyle, 2022). In response to these challenges, the Plannier event management application was developed with the goal of helping users in handling event-related tasks, such as managing to-do lists and sending invitations. Furthermore, Plannier provides a method for users to capture and store event memories by converting images into videos.

The Plannier application is built using Flutter, an open-source UI software development kit created by Google. Flutter simplifies the process of building cross-platform applications by providing a consistent framework and utilizing the Dart programming language. However, the Flutter community faces limitations when it comes to accessing native APIs and utilizing specific features of the native platforms, especially for image-to-video conversion.

This bachelor thesis focuses on the development of a plugin within the context of the Plannier application. In particular, this thesis explores the creation of a plugin that enables image-to-video conversions. The motivation for developing the plugin results from the need for a more efficient and effective solution for image-to-video conversions within the Flutter framework. Existing options rely on outdated APIs, slowing down performance and limiting capabilities. The purpose of creating a new plugin, the FlutterMediaWriter Plugin, is to access the latest native APIs and provide users with a simplified process for converting event photos into videos.

The manual conversion of images to the YUV_420_888 color space is the main focus of this thesis's technical investigation of the image-to-video conversion procedure. We obtain a greater understanding of the technical aspects involved by comprehending the principles of the conversion process and working with the fundamental components.

Throughout this thesis, we will assess the impact and benefits of utilizing the YUV_420_888 color space for efficient image compression and video format compatibility.

By examining the plugin development process and investigating the image-to-video conversion technique, this thesis wants to contribute to the advancement of plugin development in Flutter.

# 2. Theoretical Background

## 2.1. Context

Flutter is a cross-platform technology that has gained recognition for its ability to create applications with native-like user interfaces on both iOS and Android platforms. However, it has faced a major challenge when it comes to converting images to videos. Despite its many advantages, Flutter does not have a built-in library or package that provides an efficient way for image-to-video conversion. This limitation is problematic for those who want to store their memories in an easily accessible format, hence the idea for this Bachelor Thesis.

To address this issue, we developed a plugin for Flutter that enables users to convert images to videos with ease. The plugin uses the native methods for image-to-video conversion on each platform. For Android, we utilized Media Muxer and for iOS, we used AVAssetWriter.

The only other solution we had for image-to-video conversion in Flutter beforehand was the FFmpeg library. FFmpeg is a powerful tool for multimedia processing. In general, FFmpeg provides more flexibility and control over the image-to-video conversion process compared to Media Muxer, which is a simpler and more straightforward option. The trade-off is that FFmpeg can require more processing power and memory compared to Media Muxer, which can lead to slower performance in some situations. Additionally, FFmpeg receives JPGs files as input for the image-to-video conversion and for our alternative, we converted the JPGs to the YUV_420_888 format which is more suited for video processing because it provides a more direct representation of the color information in an image, allowing for more efficient processing of the image data. The use of YUV in image-to-video conversions results in smoother and more accurate video playback compared to JPG.

In conclusion, the solution that we have developed is more suitable for the purpose of the Plannier application as it leverages the use of native methods, ensuring that the image-to-video conversion process is performed in an optimized manner with a focus on speed and accuracy of image processing.

## 2.2. Conversion to YUV_420_888

The YUV color space is a color format that is widely used in video processing and transmission. This color space is composed of three components: Y, U, and V. Y stands for "luma," which represents the brightness or lightness of the image. U and V represent the "color difference" signals, with U representing the blue minus luma (B-Y) and V representing the red minus luma (R-Y) color information.

The YUV color space is particularly advantageous for video compression and transmission because it separates color information from brightness data. This separation can reduce file sizes and make video streaming more efficient. Additionally, the YUV color space is widely supported by hardware and software video codecs, making it an ideal choice for video processing and transmission. Another benefit of using YUV is that it requires less processing power than other color spaces like RGB, which can be computationally intensive. Finally, because YUV separates luminance and chrominance information, it can result in better color accuracy compared to other color spaces.

In Flutter, to prepare an image for video encoding, it must be converted into YUV format. This is done by first converting the image into an array of bytes in RGBA format. The bytes are then processed using a formula that converts RGB values into YUV values. The formula for converting RGB into YUV is the folowing:

Y = 0.299 x R + 0.587 x G + 0.114 x B + 0

U = -0.169 x R-0.331 x G + 0.499 x B + 128

V = 0.499 x R-0.418 x G-0.0813 x B + 128

(*Audio and Video Codec: YUV444, YUV422, YUV420*, 2020)

The Y component represents brightness, while the U and V components represent color information. The Y, U, and V components are stored in separate arrays, with each array containing all the values for that particular component. The YUV format offers several advantages over other color spaces, including better video compression and better color accuracy.

During the conversion process, four bytes are extracted at a time from the RGBA array, but only the first three (RGB) are used in the conversion, as seen in Figure 2.1 at line 56-58. The alpha channel is not taken into account in the conversion process since it is based solely on the red, green, and blue

channels. Once the Y, U, and V components have been calculated for each set of RGB values, they are stored in their respective arrays in YUV444 format. This process is repeated until all the RGB values in the image have been processed.

```dart
static Uint8List convertRgbToYuv444(Uint8List pixels, int width, int height) {
  final int ySize = width * height;
  final Uint8List yuvy = Uint8List(ySize);
  final Uint8List yuvu = Uint8List(ySize);
  final Uint8List yuvv = Uint8List(ySize);

  int index = 0;
  for (int i = 0; i < pixels.length; i = i + 4) {
    int r = pixels[i + 0];
    int g = pixels[i + 1];
    int b = pixels[i + 2];

    final int yValue = (0.299 * r + 0.587 * g + 0.114 * b).round();
    yuvy[index] = yValue.clamp(0, 255);

    int uValue = (-0.147 * r - 0.289 * g + 0.436 * b + 128).round();
    yuvu[index] = uValue.clamp(0, 255);

    int vValue = (0.615 * r - 0.515 * g - 0.100 * b + 128).round();
    yuvv[index] = vValue.clamp(0, 255);
    index++;
  }
  var yuv1 = convertYUV444toYUV420(yuvy, yuvu, yuvv, width, height);
  return yuv1;
}
```

Figure 2.1. Conversion of RGB Array to YUV444 Format

In the YUV444 format, the pixel data is represented as Y0 U0 V0 Y1 U1 V1 Y2 U2 V2. This format preserves all the pixels that were converted during the process.

Finally, the YUV444 arrays are converted into YUV420 format. This involves downsampling the U and V components to reduce the amount of data needed to store the image.

Experts have found that the data of two pixels next to each other in a row, as well as the data of two pixels in the same position in the next row, are very similar. This means that we can use downsampling to potentially remove some of this redundant data. (*Audio and Video Codec: YUV444, YUV422, YUV420*, 2020). After conversion the format is named YUV420.

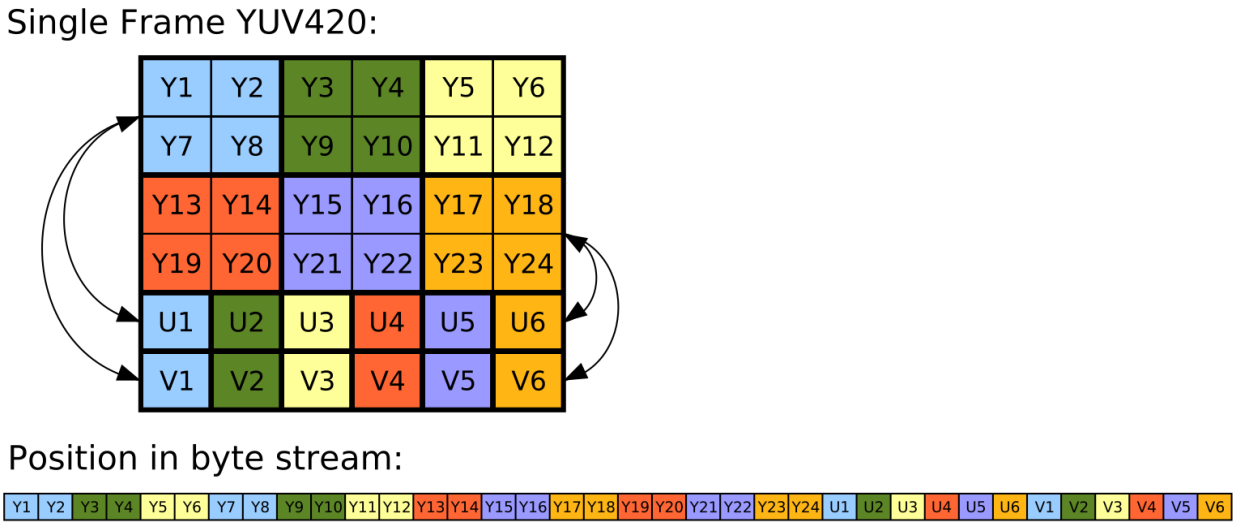



Figure 2.2. Representation of the bytes in YUV420 format. (*YUV*, 2015)

In the YUV420 format, the number of pixels differs from the YUV444 format. Instead of storing each pixel's RGB information separately, the number of pixels is determined using the formula *width* * *height* + *width* * *height* / *2*. This means that in YUV420, only a quarter of the U component and a quarter of the V component are stored compared to the length of the Y component, which is stored completely. This reduction in size is possible because experts, as mentioned before, have discovered that the data of two neighboring pixels in a row, as well as the data of two pixels in the same position in the next row, are very similar. Therefore, we can take groups of four pixels, as presented in Figure 2.2, calculate their average values, and store the result.

This conversion process is illustrated in Figure 2.3. Firstly, the lengths of the Y, U, and V bytes are determined based on the dimensions of the image. The Y component is stored in its entirety without any modifications.

Next, the U component undergoes downsampling. The U data is grouped into sets of four adjacent bytes, representing four neighboring pixels. For each group of four U bytes, their values are averaged, resulting in a single average value. This average value represents the U component for that group of pixels. This downsampling process is performed for all groups of four U bytes in the image or video.

Similarly, the V component also undergoes downsampling. The V data is grouped into sets of four adjacent bytes, and the values of these bytes are averaged to obtain a single average value. This

average value represents the V component for that group of pixels. This downsampling process is applied to all groups of four V bytes in the image.

```dart
static Uint8List convertYUV444toYUV420(Uint8List yData, Uint8List uData,
    Uint8List vData, int width, int height) {
  final yLength = width * height;
  final uvLength = yLength ~/ 4;

  final yuv420 = Uint8List(yLength + 2 * uvLength);

  yuv420.setRange(0, yLength, yData);

  for (var y = 0; y < height; y += 2) {
    for (var x = 0; x < width; x += 2) {
      final uIndex = y ~/ 2 * width ~/ 2 + x ~/ 2 + yLength;
      final vIndex = uIndex + uvLength;
      yuv420[uIndex] = (uData[y * width + x] +
              uData[y * width + x + 1] +
              uData[(y + 1) * width + x] +
              uData[(y + 1) * width + x + 1]) ~/
          4;
      yuv420[vIndex] = (vData[y * width + x] +
              vData[y * width + x + 1] +
              vData[(y + 1) * width + x] +
              vData[(y + 1) * width + x + 1]) ~/
          4;
    }
  }

  return yuv420;
}
```

Figure 2.3. Conversion of YUV444 into YU420

The purpose of downsampling is to decrease the size of the video without significantly affecting its perceived quality. This is because the human eye is more sensitive to changes in brightness than changes in color (Iowa State University, 2023). By reducing the resolution of the color information, we can achieve a smaller file size without a noticeable loss in quality.

In Figure 2.2, we can see the arrangement of the pixels in the byte stream after all the processing is complete. The pixels are stored in the order of Y, U, and V. When the pixels are stored in this order, the format is named YUV420 planar.

Once the conversion to YUV420 is complete, the resulting byte array for the current picture is sent to the FlutterMediaWriter plugin.

Each selected photo goes through the process of being encoded into the YUV420 format before

being passed to the FlutterMediaWriter Plugin for further encoding into an mp4 file. Once all the photos have been encoded as YUV420 and the YUV420 arrays have been encoded into the mp4 file, the resulting video is saved onto the device. This allows us to store videos in a more efficient manner without compromising the visual experience.

## 2.3. FlutterMediaWriter Plugin

The FlutterMediaWriter Plugin is the plugin made for the image-to-video conversion. It utilizes the native APIs in order to bring into Flutter only the functionalities needed for this application, namely, the image-to-video conversion from YUV to mp4.

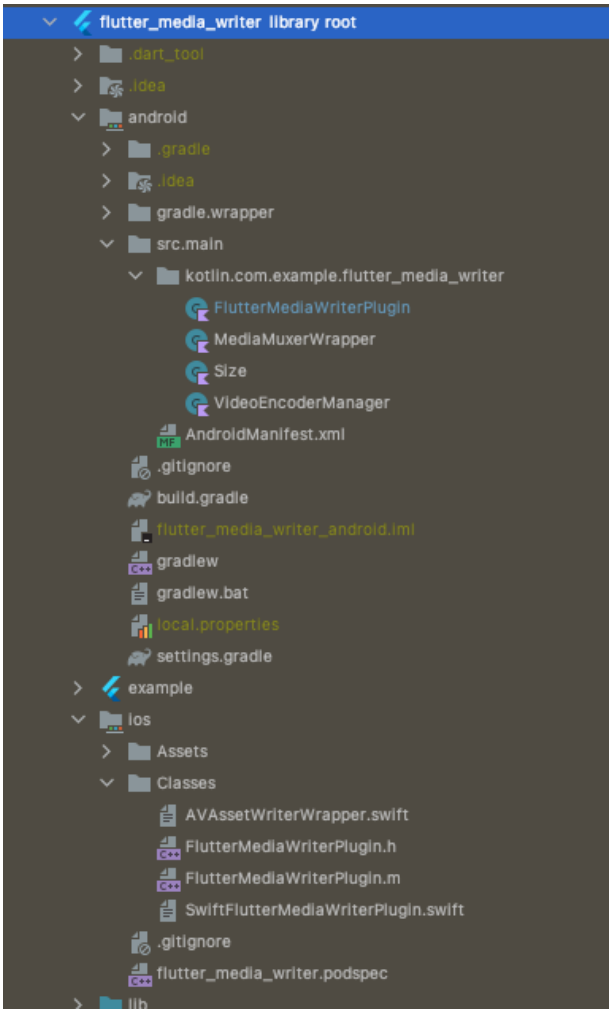



Figure 2.4. Plugin project structure

Among Flutter's great features is also the ability to use plugins to extend the functionalities that can be developed. Plugins allow developers to access device-specific APIs or add custom functionality to their apps. Flutter has an extended documentation about the creation of a plugin, which is very

helpful for developers. The creation of a plugin in Flutter is made from the terminal by running the instruction “flutter create --template=plugin <plugin-name>”. Then the platforms must be specified and after that the implementation can start. The developer has to write the plugin in both the platform-specific and Dart code.

The plugin code is written separately for the Android and iOS platforms, residing in their respective directories within the project structure, as seen in Figure 2.4. The Android code is located in the Android folder, while the iOS code is located in the iOS folder.

The Android folder contains the classes specific to the Android platform, which handle the functionalities and interactions required by the plugin. The handle method is in the FlutterMediaWriterPlugin and the actual encoder is in the VideoEncoderManager class. Similarly, the iOS folder contains the classes specific to the iOS platform, responsible for implementing the plugin's functionalities on iOS devices. The handle method is also in the FlutterMediaWriterPlugin file and the actual implementation of the encoder is in the SwiftFlutterMediaWriter file.

By organizing the code in separate directories, it allows developers to focus on platform-specific implementations, ensuring compatibility and optimal performance on each platform. This separation also enables easier maintenance and updates for individual platform requirements.

After writing the code for the plugin, the developer has to create a Flutter interface that can call the platform-specific code using the platform channels. (*Developing Packages & Plugins | Flutter*, 2017)

```
15
16    Future<String> prepare(String outputPath, int width, int height) async {
17      final args = <String, dynamic>{
18        'outputPath': outputPath,
19        'width': width,
20        'height': height,
21      };
22      return await _methodChannel.invokeMethod('prepare', args);
23    }
24
```

Figure 2.5. Prepare method in FlutterMediaWriter

FlutterMediaWriter is the class which invokes the methods for the platform-specific code. There are a few methods that are needed for the conversion to mp4, namely prepare, encode and stop. These

are the methods that the user can call from Flutter when using this plugin. As an example, in Figure 2.5, the method prepare is presented.

Both Android and iOS platforms have a class responsible for managing the communication between Flutter and the native code. In Android, this class includes a method called onMethodCall, as shown in Figure 2.6. The purpose of this method is to handle the incoming method calls from the FlutterMediaWriter class.

Within the onMethodCall method, the method name is checked to determine the appropriate action to take. It is important to ensure that the method names used in Flutter match those expected in the onMethodCall method of the native code. This ensures that the correct functionality is triggered and the desired operations can be performed seamlessly.

Additionally, the method arguments from Flutter are passed as a map object, containing key-value pairs, as seen in Figure 2.5. Therefore, in the onMethodCall method, it is crucial to use the same


Figure 2.6. Handle method for Android

keys to retrieve the corresponding argument values. This way, the native code can properly recognize and process the arguments sent from Flutter.

```
public func handle(_ call: FlutterMethodCall, result: @escaping FlutterResult) {

    if (call.method.elementsEqual("prepare")){
        guard let arguments = call.arguments as? [String: Any],
              let outputPath = arguments["outputPath"] as? String,
              let width = arguments["width"] as? Int,
              let height = arguments["height"] as? Int else {
            result(FlutterError(code: "Arguments error prepare", message: "Missing or invalid arguments", details: nil))
            return
        }
        result(avAssetWrapper.prepare(outputPath: outputPath, width: width, height: height)  )
    }
    else
    if (call.method.elementsEqual("encode")){
        guard let arguments = call.arguments as? [String: Any],
              let frame = arguments["byteBuf"] as? FlutterStandardTypedData,
              let data = Data? (frame.data)
              else {
            result(FlutterError(code: "Arguments error encode", message: "Missing or invalid arguments", details: nil))
            return
        }
        result(avAssetWrapper.encode(frame: data))
    }
    else
    if (call.method.elementsEqual("stop")){
        avAssetWrapper.stop()
        result("hello")
    }
}
```

Figure 2.7. Handle method for iOS

In iOS, similar to Android, the handle method, presented in Figure 2.7, plays a vital role in handling method calls originating from Flutter. It checks the method name to determine the appropriate action to be taken, just like the onMethodCall method in Android. It is essential to ensure that the method names used in Flutter match those expected in the handle method of the native iOS code, just as it is crucial in Android. This consistency ensures that the correct functionality is triggered, enabling seamless execution of the desired operations in both platforms.

Furthermore, similar to Android, the method arguments from Flutter are passed as map objects, containing key-value pairs. These maps allow the necessary data to be transmitted between Flutter and the native code. It is equally important in both platforms to use the same keys when retrieving the corresponding argument values in the handle methods of the native code.

As for the implementation of the conversion, for both platforms, prepare is a method that receives the path where the video should be saved and the width and height of the images.

### 2.3.1. Android implementation

In Android, the start method is called on the encoder class to initiate the encoding process. This method first creates a format for the output file based on the width and height of the pictures. Next,

it creates a MediaCodec object of type MediaFormat.MIMETYPE_VIDEO_AVC, which is specifically used for H.264 encoding.

Once the MediaCodec is created, the onInputBufferAvailable and onOutputBufferAvailable callbacks are set on it. These callbacks are used to receive notifications for the availability of input and output buffers during the encoding process. When the onInputBufferAvailable callback is invoked, it means that an input buffer is ready to be filled with the next segment of raw media data for processing. The getInputBuffer method of the MediaCodec class is typically used to access the input buffer in order to provide the data.

On the other hand, the onOutputBufferAvailable callback is triggered when an output buffer is available for consumption. Output buffers contain the processed media data after encoding or decoding. Once this callback is received, we can retrieve the processed data from the output buffer and perform other operations, such as writing it to a file. To access the output buffer, we would use the getOutputBuffer method of the MediaCodec class.

Finally, after setting up the callbacks, the start method starts the MediaCodec, initiating the actual encoding process of the file into the H.264 format. H.264 is widely used due to its highly efficient compression algorithm, which significantly reduces the size of video files while maintaining high-quality video output.

The encode method receives the YUV420 array byte that was converted before and passes it to the encode method from the VideoEncoderManager class. The method from the VideoEncoderManager class is responsible for encoding a frame of raw media data into the H.264 format. It takes a frameData parameter, which represents the raw media data in the form of a byte array.

Inside the method, it first checks if the frameData is null. If it is null, the method returns without further processing.

If the frameData is not null, the method calls the queueInputBuffer method with the frame data for encoding as parameter. This method handles the queuing process. It obtains an available input buffer from the MediaCodec using inputBuffersQueue.take() to retrieve the next available index of an inputBuffer. The inputBuffer is accessible by using the method getInputBuffer(index). Then it puts the frameData into the buffer using the put method. The input buffer is then queued to the MediaCodec for encoding using the queueInputBuffer method.

Once the input buffer is queued, the encoding process starts asynchronously. The MediaCodec processes the input buffer and generates encoded data. When the encoding process is done, the MediaCodec invokes the onOutputBufferAvailable callback of the MediaCodec.Callback interface.

The onOutputBufferAvailable callback is responsible for handling the availability of output buffers that contain the encoded data. In this callback, the dequeueInputBuffer method is called. This method retrieves the output buffer from the MediaCodec using the index that the onOutputBufferAvailable callback provides. It checks if the output buffer is valid and not null. If the buffer is valid, it proceeds to write the encoded data into an MP4 file using the mediaMuxer.writeSampleData method.

The dequeueInputBuffer method also checks the bufferInfo parameter, which contains information about the encoded data, such as its size and flags. If the buffer contains the flag BUFFER_FLAG_CODEC_CONFIG, it means that the current output buffer corresponds to codec configuration data, rather than regular encoded frames, hence it releases the output buffer without further processing. Otherwise, if the buffer contains valid encoded data, it writes the data to the MP4 file using the MediaMuxer by calling the writeSampleData method.

The encoding process continues until all frames have been processed. Finally, when the encoding is complete, the onOutputBufferAvailable callback receives the BUFFER_FLAG_END_OF_STREAM flag, indicating that all the frames have been encoded. In this case, the MediaMuxer is stopped and released.

The last method that is used for this conversion is the stop method. It doesn't receive any parameters and it calls the stopEncoder method in the VideoEncoderManager class. The stopEncoder method is called to stop the video encoder and release any allocated resources. It begins by resetting the outputVideoIndex variable to its initial value of 0. Then, it checks if the mediaCodec instance is not null and if the media codec has been started (isMediaCodecStarted flag is true). If these conditions are met, it proceeds to stop the media codec by calling its stop method, which stops the encoding process. Next, it releases the media codec by calling its release method, which frees up system resources associated with the codec.

After stopping and releasing the media codec, the method checks if the mediaMuxer instance is not null. If it is not null, it means that a MediaMuxer object was created for writing the encoded frames to an MP4 file. In this case, the method calls the stop method of the MediaMuxer to indicate that no

more data will be written to the file, and then releases the MediaMuxer resources by calling its release method.

Finally, the method clears the inputBuffersQueue to ensure that any pending input buffers are removed.

### 2.3.2. iOS implementation

In the iOS implementation of the FlutterMediaWriter, the AVAssetWriter class is utilized for encoding the video frames. The methods which are called are similar because consistency is important for plugin development.

The first method is prepare, which is a method in the AVAssetWriterWrapper class that initializes the video encoding process using the AVAssetWriter framework in iOS. It takes the output path, width, and height of the video as input parameters.

First, the method initializes an instance of AVAssetWriter with the output URL and sets the file type to .mp4. Next, the method sets up the settings assistant dictionary with the video encoding settings, including the video codec type as h264 (the same as for Android), the width, and the height.

After that, the source pixel buffer attributes dictionary is defined. It specifies the pixel format type, width, height, and bytes per row alignment. The sourcePixelBufferAttributesDictionary is used to specify the attributes and properties of the pixel buffer that will be used to store and process the video frames during encoding.

Using the specified media type of .video, an instance of AVAssetWriterInput is created with the output settings set to the settings assistant dictionary.

To facilitate pixel buffer management, an instance of AVAssetWriterInputPixelBufferAdaptor is created, taking the asset writer input and the source pixel buffer attributes dictionary as parameters. The AVAssetWriterInputPixelBufferAdaptor acts as an intermediary between the pixel buffer containing the video frame data and the asset writer.

The asset writer input is then added to the asset writer to receive video data.

Subsequently, the video writing process is initiated by calling startWriting() on the asset writer.

Finally, to begin the writing session at the source time CMTime.zero, the method calls

startSession(atSourceTime:) on the asset writer. By passing CMTime.zero as the argument to startSession(atSourceTime:), we are instructing the asset writer to start the session at the beginning of the source time. In other words, it marks the starting point of the encoding session at the very beginning of the video timeline.

In the encode method, the primary goal is to encode the frame data and write the frames to the output file using the AVAssetWriter functionality.

Firstly, a CVPixelBuffer is created to be used as a buffer for storing the pixel data of the frame. This buffer provides a suitable format for the encoding process.

Next, the base address of the pixel buffer is locked to ensure safe access and manipulation of its contents. This allows for efficient copying of the Y (luminance) and UV (chrominance) plane data from the YUV420 array into the pixel buffer.

The Y plane data is copied into the pixel buffer, followed by the UV plane data. These operations ensure that all essential pixel information is accurately transferred to the buffer.

Once the pixel buffer is accurately populated, the base address is unlocked, enabling further processing of the frame data. At this point, the method checks if the assetWriterInput is ready to receive more media data, ensuring a smooth writing process.

If the assetWriterInput is ready, a frame time is calculated using the frameCount and the desired frames per second. This frame time represents the exact presentation time of the current frame.

Finally, the pixel buffer, along with its corresponding presentation time, is appended to the assetWriterAdaptor, a pixel buffer adaptor associated with the assetWriterInput. This ensures that the frame data is written to the output file, Maintaining the original sequence and timing.

The stop method is responsible for finalizing the encoding process and concluding the writing of frames to the output file. It ensures that all pending operations are completed and prepares the output file for further processing or playback. This method marks the input as finished by using the markAsFinished method, and resets the frameCount variable. The stop method ends the encoding process and ensures the output file is complete and error-free.

# 3. Technologies

## 3.1. Flutter

Flutter is an open-source UI software development kit, developed by Google, first released in 2017. It allows developers to create high-quality, native mobile apps for both Android and iOS platforms with a single codebase.

One of the key features of Flutter is its widgets, which are building blocks for creating user interfaces. Flutter provides a vast collection of customizable widgets, including text, buttons, images, and more, which can be combined to create complex and beautiful UIs. Widgets in Flutter are also fast and efficient, allowing for smooth animations and transitions.

Flutter also offers a hot reload feature, which is a powerful tool for app development. With hot reload, developers can make changes to their code and see the results immediately, without needing to rebuild the entire app. Flutter does not redraw all the widgets on the screen when hot reload is used. It only recompiles the code that has changed and replaces the existing widget tree in the application with a new widget tree based on the updated code. For example, if the text of a button in the application is changed, Flutter would only redraw that button and any other widgets that depend on the button's state. This feature allows for a faster development cycle and more efficient debugging, although, in some cases when complex changes have been made, it may require a full application restart in order to be able to see the changes.

Another significant advantage of Flutter is its cross-platform development capabilities. Flutter enables developers to create applications for both Android and iOS platforms using the same codebase. This approach reduces development time, costs and effort in developing and maintaining separate applications for both platforms.

Flutter also has a vibrant and supportive community, with many resources and tools available to developers. There are numerous packages and plugins available in the Flutter ecosystem, which can help developers speed up the development process and add functionality to their applications.

Flutter's performance is one of its significant advantages. It provides a high-performance engine, which allows for faster app startup times, smoother animations, and faster rendering of complex interfaces. This performance is due to the fact that Flutter compiles to native code, making it more performant than other cross-platform solutions that rely on JavaScript or other interpreted languages.

While there are many advantages to using Flutter, there are also some disadvantages that should be taken into consideration before using this framework. Among the disadvantages of using Flutter are limited third-party libraries, application size and native integration.

The first one refers to the fact that, because Flutter is a relatively new technology, it has a limited number of third-party libraries, which means that for some functionalities, the developers may need to write the code from scratch. This is the case with Plannier, where the image-to-video conversion functionality was needed, which could not be developed efficiently with the resources that are already available. Hence, the need to make a Plugin from scratch, which uses the native APIs.

Another disadvantage that was mentioned is the application size. Flutter applications tend to be larger than native applications, which can be a concern for users with limited storage space on their device. The main reason why Flutter applications can be larger than native applications is that Flutter requires a runtime to be included in the app package. This runtime includes the Flutter framework and the Dart virtual machine, which is necessary to run the Dart code used to build the app. Native applications, on the other hand, are compiled to machine code and do not require a runtime to be included in the app package. This can result in smaller app sizes, as only the code and resources necessary for the app's functionality are included in the package.

Lastly, when it comes to native integration, Flutter also has some limitations compared to native development. While in native development, the developer has access to all the functionalities that are available in the API that is used. In Flutter, it can be difficult to create libraries that are optimized for both Android and iOS. Each platform has its unique set of APIs, which means that developers need to write different code for each platform. Also, when developing Flutter applications, the developers use the third-party libraries that are available for accessing native functionalities, which we already discussed that are limited and with limited accessibility.

## 3.2. Dart

Dart is a client-optimized language for developing fast apps on any platform. Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks. (Dart Development Team, 2012)

Dart is a general-purpose programming language that was developed by Google in 2011. It is an object-oriented language that features a modern syntax, powerful features, and is optimized for building web, server, and mobile applications and uses a reactive-style programming model to create dynamic and performant mobile applications. Dart is a versatile programming language that

can be used for a variety of purposes beyond building mobile applications with Flutter. For example, Dart can be used for server-side development. This is made possible by the Dart VM, which allows Dart code to run natively on a server without the need for a browser.

One of the key advantages of Dart is its productivity. It was designed to be easy to learn and use, and includes features such as optional typing and asynchronous programming, which can help make development faster and more efficient. Dart also includes a large set of standard libraries, making it easier for developers to build applications quickly.

Dart is also designed to be scalable, making it a great language for building large, complex applications. It includes features such as modular development, which allows developers to split their code into smaller, more manageable parts. This makes it easier to maintain and manage large codebases over time.

In terms of syntax, Dart has a C-like syntax, making it easy for developers who are already familiar with languages like C++, Java, or JavaScript to pick up. It also includes features such as strong type checking, which can help catch errors early in the development process.

As we can see, Dart is useful for more than just Flutter applications, but in this project, it is used only as the programming language for Flutter.

### 3.3. BLoC

BLoC comes from Business Logic Component and is an architecture pattern used in Flutter to manage the state of an application. “BLoC makes it easy to separate presentation from business logic, making your code fast, easy to test, and reusable.” (BLoC Development Team, 2020).

In BLoC, the presentation layer is responsible for rendering the UI and handling user interactions, while the business logic layer handles the application's state and data. The two layers communicate through streams, which are a sequence of asynchronous events that can be used to pass data between the layers.

BLoC architecture is based on four main components: the view, the bloc, the events and the states and may possibly have a data layer or repository from where it receives the data it needs for the UI.

The view is the UI layer of the application. It is responsible for rendering the user interface and handling user interactions. This is the layer that the user interacts with directly.

The Bloc is the business logic layer of the application. It manages the state of the application and

provides data to the view layer. When an interaction has taken place in the view layer, an event is sent to the bloc layer, which processes it and emits new states to the view layers containing the information that are needed according to the event that happened.

The event layer is the one responsible with catching the interaction that the user has with the application. These events are the inputs that are generated by the view layer and are passed to the bloc layer to be processed. For example, an event would happen when the user presses a button or submits a form.

The states layer is the one responsible with providing information to the UI layer from the bloc layer. The current state of the application is stored here, which can be used to update the UI.

An example of how the BLoC is used in an application is provided through Figure 3.1. The user interacts with the UI and, in this way, it creates an event. The event triggers the business logic layer which processes the event by requesting data from an API or a repository and then emits a new state with the updated information that was received from the data layer. Finally the view layer receives the new state and updates the UI accordingly.

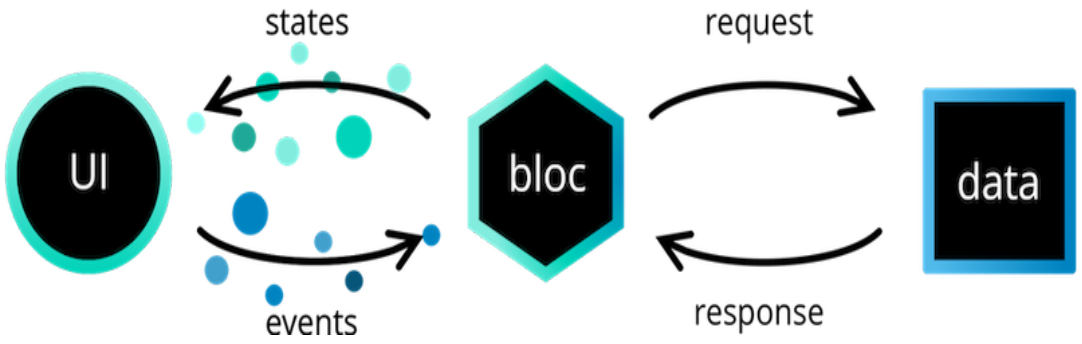



Figure 3.1. BLoC cycle in an application (*Bloc | Dart Package*, 2020)

BLoC has many advantages when it comes to using it with Flutter, because it has a lot of resources made by Flutter to help the developers with the implementation of this architecture in their project. The documentation that exists contains easy-to-understand explanations and a lot of examples to help with a deeper understanding of the concepts underlying the BLoC architecture. On top of that, BLoC has developed a package specially made for flutter, which is called flutter_bloc. This package makes implementing the architecture possible and smooth for Flutter developers. Other advantages that BLoC brings to the table are reusability, asynchronous operations and scalability.

The reusability refers to the fact that BLoC components can be easily reused in different parts of the

application. BLoC also supports asynchronous operations which allows the application to run smoothly and efficiently when performing multiple operations at once. Lastly, the scalability means that the architecture can handle large and complex applications with ease.

Although BLoC has many advantages, there are also disadvantages to take into consideration when working with BLoC. For example, BLoC is suitable for larger applications. Because BLoC is a complex architecture pattern, it can be overkill for the small applications by adding unnecessary complexity. In its implementation, it also requires a lot of boilerplate code, which can be time consuming and less efficient in the development process.

Another alternative to BLoC architecture may be the Provider. It is a simple, lightweight architecture pattern that can be used to manage the state of a Flutter application. It is easier to learn and implement than BLoC, and is better suited for smaller applications.

In conclusion, BLoC is a powerful architecture pattern that is well-suited for managing the state of large and complex applications in Flutter. Its separation of concerns, reusability, support for asynchronous operations, and scalability make it a popular choice among developers. However, it can be difficult to learn and implement, and may be overkill for smaller applications.

### 3.4. Firebase

Firebase is a mobile and web application development platform, developed by Google, that provides a variety of backend services, including authentication, real-time database, cloud storage, cloud messaging, and more. It offers a convenient and efficient way to develop and manage mobile and web applications, and it can be integrated into Flutter apps with ease.

One of the key advantages of using Firebase with Flutter is its real-time database feature. This allows developers to create applications that can synchronize data in real-time, which is especially useful for building collaborative apps like chat applications or multiplayer games. Firebase also provides a variety of authentication methods, including email and password, Google, Facebook, Twitter, and GitHub authentication, making it easy to implement user authentication and authorization.

Another advantage of using Firebase with Flutter is that it provides cloud storage for files, such as images or documents. This allows developers to store data in the cloud and retrieve it when needed, without having to worry about local storage constraints. Firebase also provides cloud messaging, which enables developers to send notifications to users across different platforms and devices.

Among other functionalities that Firebase has are cloud functions, performance monitoring and analytics. Cloud functions are a serverless framework that lets the developer run the backend code in response to events triggered by Firebase or third-party services. Firebase tracks user engagement and analyzes data to improve the application's performance and features. It also provides a detailed insight into the performance of the application, including user experience and device-specific issues. (For example, how many crashes the application had since it was released on store)

However, there are also some disadvantages of using Firebase with Flutter. One of the main concerns is that Firebase is a third-party service, and it requires an internet connection to work. This means that if there is an outage or if the connection is slow, the app may not function as expected.

Another concern with Firebase is that it can be expensive, especially for larger applications that require more storage or usage. There are also some limitations on the number of requests that can be made per second, which can affect the performance of the application.

Although it has its limitations, Firebase offers a range of services and features that are highly useful for mobile application development and it has great integration with Flutter. Firebase offers extensive documentation for integrating its services with Flutter, making it easier for developers to build real-time, data-driven mobile applications. The documentation is well-structured and easy to navigate, with step-by-step instructions, code snippets, and sample applications that help developers understand how to use Firebase with Flutter.

# 4. Application - Plannier (Events made Easy)

## 4.1. Related Work

During the research conducted, no single application with all the features encompassing both event handling and image-to-video conversion, similar to Plannier, was found. However, there are applications that excel in one of these areas. Let's explore two notable examples:

### 4.1.1. EventBrite

EventBrite is a well-known smartphone application that lets users make events and send invitations. Popular platform Eventbrite makes the process of organizing events and selling tickets easier. Both iOS and Android mobile devices support it as a mobile application.

Users of Eventbrite can plan, administer, and advertise a variety of events, from intimate get-togethers to sizable conferences. Event specifics like date, time, place, description, ticket prices, and capacity restrictions can be readily put up by users. The program offers a user-friendly interface to help users design aesthetically appealing event pages and brand them with pictures, logos, and other elements.

Once the event is created, Eventbrite offers robust invitation and ticketing features. Users can send personalized event invitations via email or social media platforms directly from the application. Additionally, Eventbrite provides tools to track invitation responses and manage attendee lists.

Moreover, Eventbrite offers seamless integration with popular calendar applications, such as Google Calendar and Apple Calendar, ensuring that users stay organized and can easily manage their event schedules.

Eventbrite and Plannier share some similarities in terms of event creation and invitation sending. Both applications aim to make event planning easier and more efficient and provide tools for managing events effectively. Both Eventbrite and Plannier share several similarities in their approach to event planning:

Firstly, both applications allow users to create events with important details such as date, time, location, and descriptions.

Secondly, both applications offer features for sending event invitations. Users can send invitations and keep track of guests’ statuses.

Furthermore, both applications integrate with popular calendar platforms such as Google Calendar and Apple Calendar. This integration allows users to sync event details and schedules seamlessly, helping them stay organized and avoid scheduling conflicts.

While Eventbrite has a broader user base and focuses more on ticketed events, Plannier places emphasis on personal event planning and offers additional features such as to-do lists. Plannier aims to assist individuals in organizing private events and managing related tasks. In contrast, Eventbrite caters to a wider range of events, including public conferences, concerts, festivals, and more.

In summary, both Eventbrite and Plannier strive to simplify event planning and offer tools for creating, managing, and promoting events. Although they have different focuses and target audiences, they share the common goal of providing users with efficient event planning solutions.

### 4.1.2. Quik - GoPro Video Editor

In the area of image to video conversion applications that are available for both Android and iOS platforms and also offered for free, the options are somewhat limited. While there are feature-rich applications like Adobe Premiere Rush available, they often come with a price tag. However, there is one exception that stands out: Quik - GoPro Video Editor. Quik is a video editing application developed by GoPro, primarily known for its action cameras. It allows users to convert their images into videos by combining them with video clips and applying basic editing features. With Quik, you can add transitions, text overlays, music tracks, and simple effects to enhance your videos.

One of the primary similarities between Quik and Plannier is their ability to convert images into videos. Both applications offer this feature, allowing users to transform their image collections into dynamic and engaging videos.

Another notable similarity is the availability of both applications on both iOS and Android platforms. This cross-platform support ensures that users can access and utilize the features of Quik and Plannier regardless of their preferred mobile operating system. This flexibility provides convenience and accessibility to a wide range of users.

Furthermore, both Quik and Plannier offer free versions of their applications. Plannier's core features are entirely free to use, providing users with event planning and management functionalities without any cost. Similarly, Quik provides a free version that includes basic editing capabilities, allowing users to start creating videos without the need for upfront payments. This free offering gives users an opportunity to explore the applications and their features before considering optional in-app

purchases or upgrading to premium versions, if available.

Plannier differentiates itself from other applications by offering a comprehensive set of features that address users' needs in event planning and image-to-video conversion. While there are applications available that provide certain aspects found in Plannier, none of them specifically focus on event planning and handling while also offering the capability to convert images to videos.

Plannier stands out by providing a comprehensive solution that combines event planning functionalities with the ability to create captivating videos from images. Unlike other applications that require users to download multiple apps for separate tasks, Plannier offers a unified platform where users can efficiently manage their events and unleash their creativity by transforming images into engaging videos. This integrated approach enhances convenience and streamlines the user experience, eliminating the need for additional downloads and ensuring a seamless and efficient workflow. With Plannier, users can effectively plan their events and unleash their multimedia creativity without the hassle of juggling between different applications.

## 4.2. Architecture

The BLoC architecture is a type of software design used for mobile and web apps, especially with Flutter. It is a variation of the Model-View-Controller design, but it adds an extra layer called Business Logic Component (BLoC).

This application utilizes the BLoC architecture pattern to manage its features. Almost every important feature in the application has a separate BLoC dedicated to it. The BLoCs created for this application include AuthBloc, EventBloc, InvitationBloc, ProfileBloc and ToDoBloc.

Each BLoC is named after the feature it manages, allowing for clear identification of its purpose. The AuthBloc manages authentication, the EventBloc manages events, the InvitationBloc manages invitations, and the ToDoBloc manages to-do lists. The BLoC architecture consists of three classes: the Event class, the State class, and the Bloc class. The Event class is an abstract class that extends Equatable, a library that helps with overriding the equal operator (==). This means that in the props method, which needs to be implemented, all the fields that need to be considered when comparing two objects of the same class must be included. The State classes, as well as all the models used within the BLoC, must also extend this Equatable class.

By using separate BLoCs for each feature, the application can maintain a clear separation of concerns and manage each feature's logic efficiently. This approach can also lead to more

maintainable and scalable code in the long run.

Essentially, the application functions based on user interactions. Each interaction prompts an event to be triggered within the application. Once the event is triggered, a corresponding method is executed within the BLoC. The method, in turn, generates a new state with the updated information that reflects the result of the user's interaction.

In order for a BLoC (Business Logic Component) to be utilized in an application, it must be created prior to being used. The process of creating a BLoC involves the utilization of the *BlocProvider(create: Bloc())* widget. If this widget is placed before the MaterialApp widget, then the resulting widget tree would contain the created BLoC. However, if the widget is placed somewhere else, then the BLoC must be provided to each individual screen that utilizes it. To avoid the repetitive creation of a BLoC each time it is needed, we can pass the value of the created BLoC to the next widget by utilizing the *BlocProvider.value(value: ...)* widget. Additionally, if we have more than one bloc and we need to pass their values to another widget, we can use a special widget called *MultiBlocProvider*, as seen in Figure 4.1. This widget enables us to wrap multiple bloc providers together so that their values can be easily accessed by the widget tree below.

```
GestureDetector(
  onTap: () => Navigator.push(
    context,
    MaterialPageRoute(
      builder: (_) => MultiBlocProvider(
        providers: [
          BlocProvider.value(value: context.read<EventBloc>()),
          BlocProvider.value(value: context.read<InvitationBloc>()),
        ],
        child: const EventPersistScreen(),
      ),  // MultiBlocProvider
    ),  // MaterialPageRoute
  ),
```

Figure 4.1. Example of MultiBlocProvider usage

When a widget within an application needs to be rerendered as a new state is emitted, the BlocBuilder widget is the recommended approach. This widget is called each time the state is updated and facilitates the efficient rerendering of widgets. For instance, when displaying a list of events, we can utilize the *BlocBuilder<EventBloc, EventState>()* widget to rerender the widget once the events have been fetched and subsequently display them. Moreover, the BlocBuilder widget provides us with the ability to control what occurs when data is loading or when an error occurs.

This facilitates the display of a widget that accurately reflects the current application state.

#### 4.2.1 AuthBloc

The AuthBloc follows the same structure as any other BLoC, which is composed of three classes: the Event class, the State class, and the Bloc class.

The AuthEvent class is an abstract class that extends the AuthEvent class and includes the following events: AuthLogin, AuthReload, AuthLogout, AuthSignup, and AuthResetPassword. These events represent the different actions that can be performed during the authentication process. Similarly, the AuthState is an abstract class that extends the Equatable class and includes the following states: Unauthenticated, Authenticated, and AuthError. These states represent the different stages of the authentication process.

As an example, let's consider the AuthSignup event, which is the most complex method in this BLoC. When a user enters the Signup screen and fills out all the necessary fields, they can press the Sign up button, which triggers the AuthSignup event by calling the *context.read<AuthBloc>().add(AuthSignup(...))* method. The details filled in by the user, such as their first name, last name, etc., are passed through this event to the AuthBloc.

In the AuthBloc, a method is created to handle this event and it uses FirebaseAuth's *createUserWithEmailAndPassword()* method to create a new user. Then, the user's details are updated with their profile picture and name using the *updatePhotoURL()* and *updateDisplayName()* methods. After that, the user is stored in the database of the project. The reference to the 'users' collection in the database is used to add a new document with all the information provided at sign up.

Finally, the state that is emitted is Unauthenticated because the user needs to log in to authenticate after signing up.

#### 4.2.2 EventBloc and InvitationBloc

The EventBloc and InvitationBloc follow the same template as other BLoCs, with Event and State classes that include events like fetch, update, add, and delete. However, what sets these blocs apart is that they communicate with each other. When a user adds an event, they select the guests they want to invite. Pressing the add or update event button triggers an EventAdd or EventUpdate event in the EventBloc. The method executed when the event is triggered stores the event data in the database, but also adds an invitation to the selected guests collection of invitations. The invitation includes the

name of the user who made the invitation and a pending response until the guest replies. After this, both blocs refetch the data to ensure that everything is up to date.

Similarly, when a guest responds to an invitation, an InvitationResponse event is triggered in the InvitationBloc. The method executed updates the user's invitation in the database and also updates the event guest list with the response. After this, the blocs refetch the data to keep it up to date.

This communication between the two blocs ensures that the events and invitations remain synchronized, and everyone stays informed about the event's status. By following this process, the blocs make it easy to manage and update events and invitations, providing a smooth user experience.

### 4.2.3 Widget Tree

One of the key concepts in Flutter is the widget tree, which is a hierarchical structure that defines the layout and behavior of widgets within an application.

In a Flutter application, the widget tree is created by nesting widgets inside each other to create a hierarchy of parent and child widgets. The parent widget controls the layout and behavior of its child widgets, and this nesting pattern continues down the tree until the leaf widgets are reached.

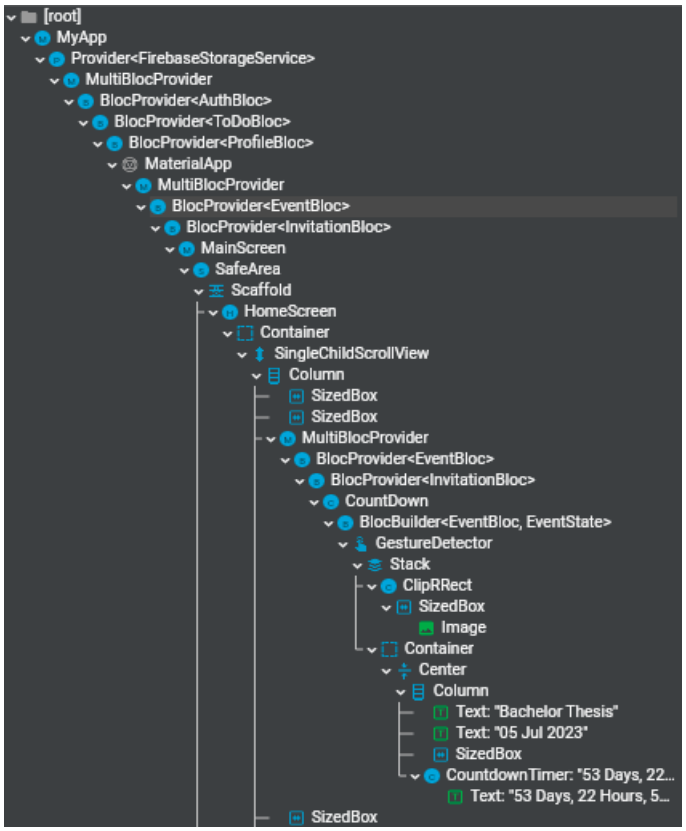



Figure 4.2. Example of widget tree

Each widget in the tree is responsible for rendering a specific part of the UI, such as a button, text box, or image. These widgets are customizable and can be configured to fit the specific needs of the application. For example, Figure 4.2, displays the widget tree from the beginning of the "My App" application to the HomeScreen. This figure shows that the blocs for Auth, ToDo, and Profile are created above the MaterialApp. This means that these blocs are accessible from anywhere in

the app without the need to pass their values to the widgets. On the other hand, the rest of the blocs are created after the MaterialApp, so we need to pass their values to every widget that needs them, as done in Figure 4.1.

## 4.3. Features

This application comes equipped with several essential features that make it a comprehensive tool for anyone looking to plan an event.

First and foremost, users can create their personal accounts with a secure sign-up and login screen. This ensures that their information is kept safe and accessible only to them, and they can easily access the application whenever they need to.

The to-do list is another essential feature of this application, which helps users keep track of everything they need to do. Users can create a list of tasks that need to be completed in order to stay organized and ensure that nothing is overlooked while planning the event.

One of the most significant challenges of planning an event is keeping the guest list up to date and accurate. This application simplifies this task by allowing users to invite other users to their event. This feature makes it easy to keep track of who has been invited, who has responded, and who has yet to reply.

Another standout feature of this application is the ability to add photos from the event and convert them into videos. This is an excellent way for users to store their memories in a more accessible and enjoyable format.

### 4.3.1 Sign up and Login Features

In the digital age, privacy and security are fundamental concerns, particularly with respect to the handling of personal information and events. To address these concerns, the Sign up and Login Feature has been integrated into the application to ensure the confidentiality and security of the

users' data and activities.

The Sign-up screen has been designed to be user-friendly and straightforward. Users are only required to enter their first name, last name, email address, and a chosen password, as seen in Figure 4.3. Additionally, they have the option to personalize their account by adding a profile picture. This picture is stored in Firebase Storage for future access. When the “Upload image” button is pressed, the user is directed to the gallery where they can choose an image. Once an image is selected, it is converted to bytes and its path is determined through the concatenation of the directory and image name. The path and image are then transferred to another method - with the image as Uint8List and the path as String - where the image is uploaded using the FirebaseStorage putData() method.

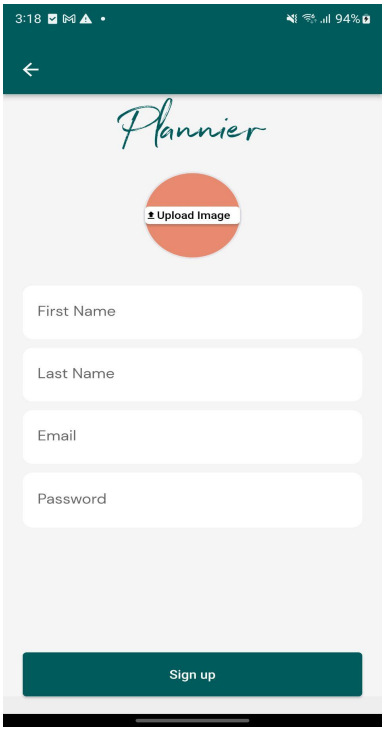



Figure 4.3. The Signup Screen

Once the upload is complete, the image URL is downloaded, in order to display the new image URL as the profile picture on the screen and store it together with the user's details in Firebase for future reference.

Upon selecting the Sign up button, the provided information is automatically verified by the Firebase Auth to ensure its validity. If the verification process is successful, the user's details are securely stored in the database.

Errors can occur during the Sign up process, and to help users overcome them, a popup has been implemented that appears regardless of the outcome. If the information provided is valid, the popup reads "Success. Login into your account to continue." In contrast, if an error has occurred, the popup clearly communicates the specific issues that require modification to successfully complete the sign up process.

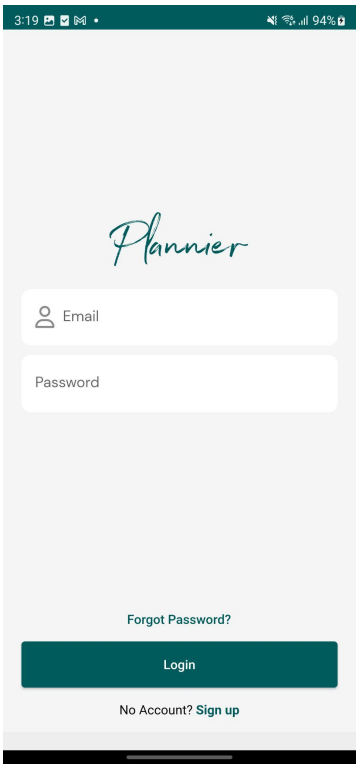



Figure 4.4. The Login Screen

For instance, if the email address provided is already associated with another account, the popup

reads "Error. The email address is already in use by another account.".

The Login Feature, which is represented in Figure 4.4, requires users to fill in two fields: Email and Password. To make it more convenient for users, there is an eye icon button located at the end of the Password field, when it is focused. This button allows users to toggle whether the password is visible while they are typing it. If the eye icon has a diagonal line crossing it, that means the password is not visible.

Once the user completes the fields, the data is checked by the Firebase auth to ensure that it is valid. If the data is not valid, a popup will appear with a message explaining the issue. For instance, if the email does not have the correct format, a popup with a message "The email address is badly formatted" will appear. Similarly, if the user enters data that does not match any existing user, the message "There is no user record corresponding to this identifier. The user may have been deleted" will be displayed. These messages are intended to assist the user in identifying and resolving any issues with their login details.

The login feature also includes a helpful option called "Forgot Password". If a user forgets their password, they can restore it easily. On the Login Screen, there is a "Forgot Password?" button located at the bottom. By clicking it, the user is directed to a new screen where they only need to fill in one field, which is the email field. If the entered email is associated with a user, a message will be sent to that email address with a link that will allow the user to create a new password for their account.

However, if the email is not associated with any account, a popup will appear with the message "There is no user record corresponding to this identifier. The user may have been deleted". Similarly, if the email is not formatted correctly, the popup will display the message "The email address is badly formatted", as seen in Figure 4.5.

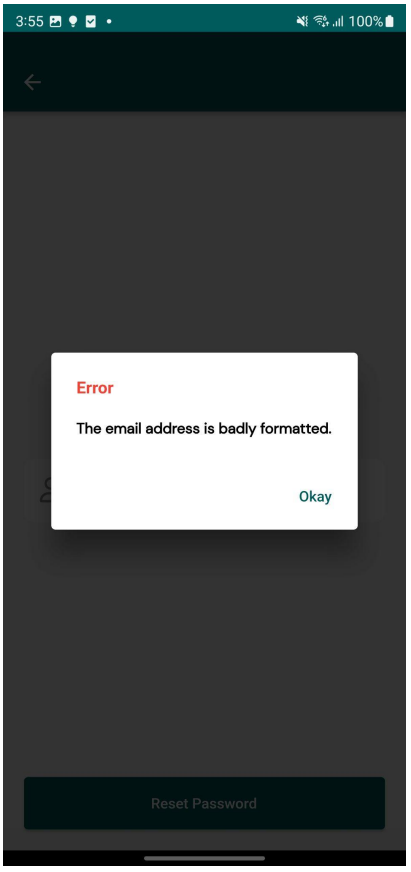



Figure 4.5. Error popup when email address is not valid

Firebase Auth provides the necessary functionalities that make these features possible.

### 4.3.2 To Do List Feature

An important feature of an event planning application is the To Do List function. This can be accessed on the third position in the BottomAppBar of the application. Users can create as many to-do lists as they need.

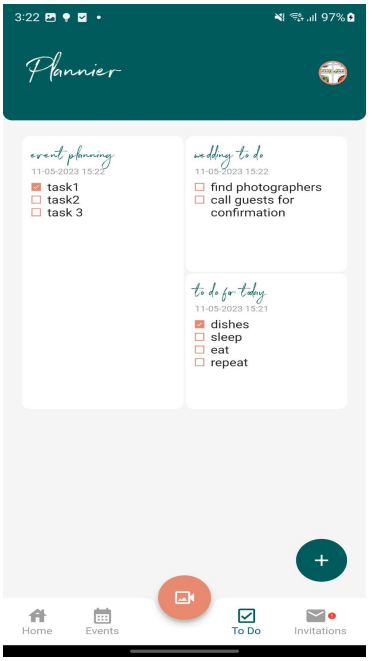
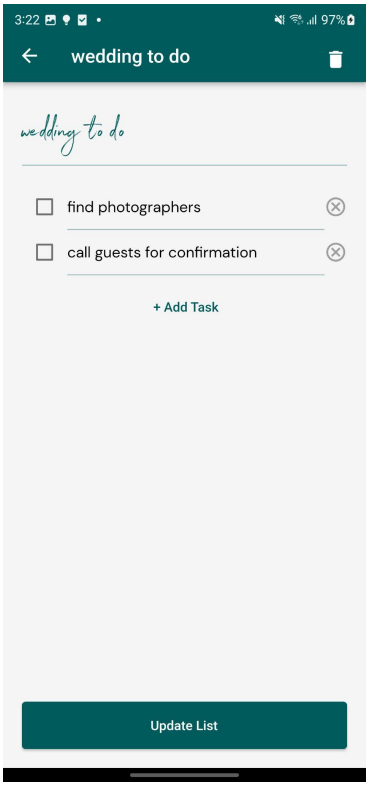



Figure 4.6. To Do Screen

To add a new list, the user needs to tap the plus button in the bottom right of the screen, presented in Figure 4.6, which opens a new screen where the user can input the list title and tasks. The task list is generated dynamically, so a new task is added to the list when the user taps the "+ add task" button. In order for the form to be valid and to save the list, the user must fill out all the fields, even the newly created field. If the user wants to remove a task field from the list, they can tap the cancel icon located at the end of each field.

The event planning application offers users the convenient option to modify their to-do lists after they have been created. This feature uses the same screen as the Add To Do List feature, which ensures that there is no duplication of code and that the code remains as reusable as possible. Each task field includes a checkbox at the beginning of the row, as seen in Figure 4.7, which enables users to track their progress and ensure that they complete all necessary tasks.




The to-do lists are ordered by the date of their last edit, which is updated every time a user modifies them in any way. This way, users can easily locate the most recent list they worked on, making their event planning more efficient. The to-do list feature of the event planning application also includes a useful option to delete a list. Users can select any previously created list and find a delete button in the app bar located at the top right corner of the screen, presented in Figure 4.7.

Figure 4.7. Edit to do list screen

When the button is pressed, a pop-up message will appear asking for confirmation before proceeding.The message will read "Are you sure you want to delete this list?" and will present the user with two options, "Cancel" and "Yes". If the user chooses "Cancel", the pop-up will close and no changes will be made. However, if "Yes" is selected, the user confirms their understanding that the list will be permanently deleted and the process will begin. The list's ID will be sent to the server to initiate the deletion process, and the data will be reloaded from the database. After the deletion process is completed, the user will be redirected to the to-do list screen, where the deleted list will no longer be visible.

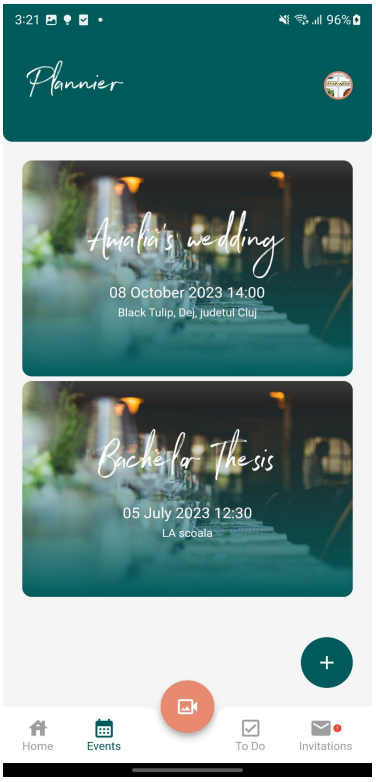



Figure 4.8. Event Screen

### 4.3.3 Events Feature

The Events feature is an important part of the Plannier Application that allows users to create and invite others to events.

This feature can be found in the second position of the BottomAppBar. When this position is selected, a screen appears, represented in Figure 4.8, containing a list of all the events the user has created.

Each item in the list displays important details, such as the title, date and time of the event, and the address, making it easy for the user to find specific details quickly.

In the bottom right corner of the screen, there is a plus button, as seen in Figure 4.8. Clicking on this button takes the user to the Add Event screen, where they must fill in all the required fields to save the event. The first field is for the title of the event. Next, there are two fields side by side for selecting the date and time of the event. These fields use a library called DateTimePicker to provide a user-friendly interface.

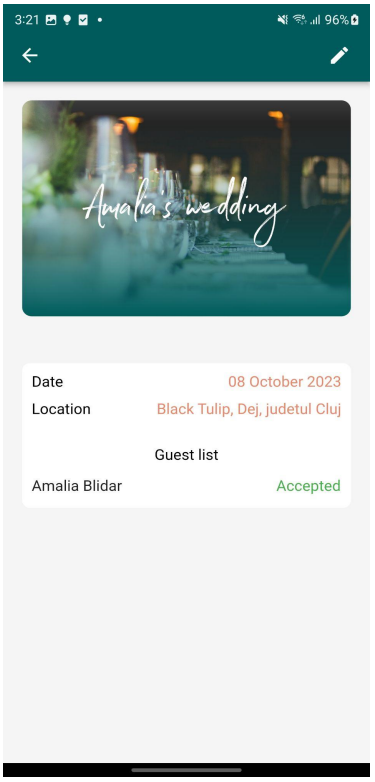



Figure 4.9. Event Details Screen

The time field displays an actual clock where the user can select the desired hours for the event. The address field must also be filled in so that guests know where the event is taking place. Lastly, there is a guest list that contains the names of all users of the application, making it easy to invite anyone who has an account. Each name on the list has a checkbox next to it so the user can see which guests have been invited. After completing all the fields, the Add Event button becomes available to be clicked. When the button is pressed, the event is stored in Firebase, and invitations are sent to all users who were checked on the guest list.

When a user clicks on an event in the list, they are taken to the event's details page, presented in Figure 4.9. On this page, the user can view the date, location, and guest list for that particular event. The invitation status for each guest is displayed next to their name.

The status can be one of three options: pending, accepted, or declined. The colors of the statuses are color-coded for easy identification of each status. Accepted invitations are displayed in green, declined invitations are shown in red, and pending invitations are displayed in gray.

The Edit Event feature in Plannier Application is located at the top right of the event detail screen and is represented by a pencil button. When the user taps on this button, they are redirected to the Edit Event screen which is similar to the Add Event screen. The difference is that all the fields are pre-populated with the existing event information, including the guest list, as seen in Figure 4.10.

The user can edit all aspects of the event, including adding or removing guests from the list. If a guest is uninvited, the checkbox next to their name can be unchecked, and the invitation will be removed from the user's invitation list.

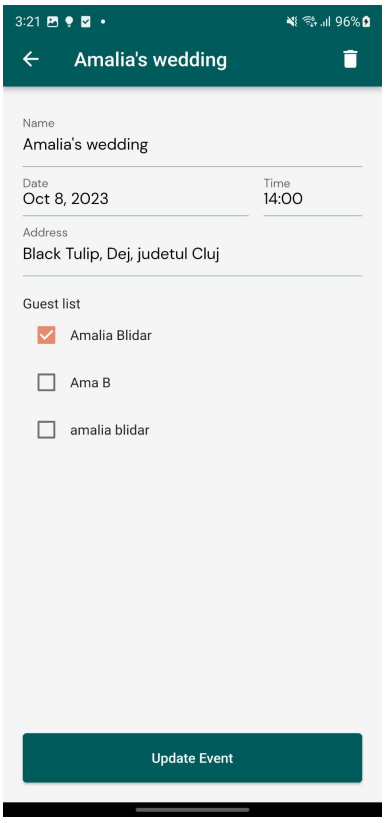



Figure 4.10. Edit Event Screen

This feature enables the event manager to make any necessary changes to the event and keep the guest list up to date.

On the editing event screen, located at the top right-hand corner of the application, a trash icon button is available for the user to delete the event. Upon clicking this button, a popup window is displayed to ensure that the user fully agrees with the action they are about to take. The popup

message displays the following question, "Are you sure you want to delete this event?" and offers two options, "Cancel" and "Yes". Selecting "Cancel" will close the popup window and return the user to the editing event screen with no further action taken. Choosing "Yes" will initiate the deletion process, whereby the "Yes" button is transformed into a CircularProgressIndicator button that spins until the process is complete. Upon completion, the user is redirected to the event list, where the deleted event is no longer visible. For the sake of maintaining a consistent visual experience throughout the application, the confirmation popup is designed to resemble other confirmation popups, such as the delete to-do list popup.

### 4.3.4 Invitation Feature

The Invitation feature in the Plannier Application is designed with a consistent appearance in order to maintain a cohesive look across the application. This feature is located fourth from the left in the BottomAppBar and provides a list of all invitations that the user has received, as presented in Figure 4.11. Each item on the list displays the event details along with two buttons: one to decline and one to accept the invitation. Upon clicking either button, a confirmation popup is displayed to ensure that the user fully understands the implications of their decision. Once the user confirms their response, the invitation's status is updated accordingly.

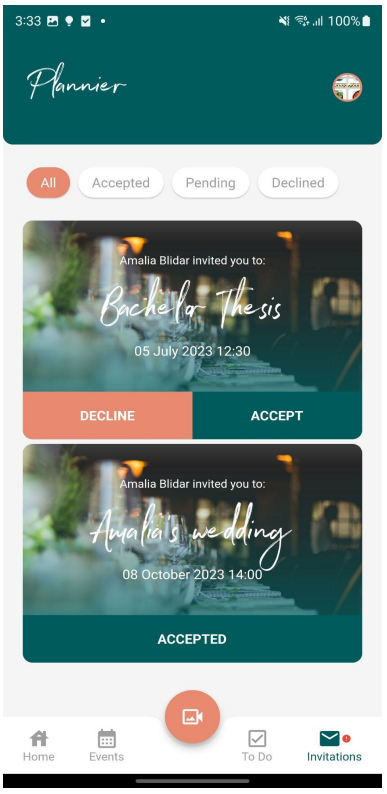



Figure 4.11. Invitations Screen

In the user's invitation screen, the buttons are replaced by color-coded cards that display the invitation's updated status. Invitations that have been declined display an orange card with the message "Declined," while those that have been accepted display a green card with the message "Accepted", as seen in Figure 4.11. These color-coded cards are implemented to enhance the list's readability.

The list is also equipped with several filtering buttons at the top, presented in Figure 4.11, which allow the user to filter the invitations by status. This feature helps ensure that no invitations are overlooked or left unanswered and also makes it easier for the user to navigate through their invitations and find the ones they are interested in. The "All" button is the default filter, which

displays all invitations, regardless of their status.

Moreover, to prevent any invitations from remaining unanswered or being overlooked, the Plannier application constantly monitors the user's invitation list for updates. When a new invitation is added, the application displays a red circle next to the invitation icon in the BottomAppBar to alert the user to its presence, as seen in Figure 4.11. The red circle is displayed continuously as long as the user has at least one invitation that remains unanswered. This ensures that the user remains informed of any pending invitations, regardless of whether they have been viewed or not.

### 4.3.5 Profile Feature

The profile feature in the application is easily accessible from every main screen as it is located on the right side of the app bar. The app bar can be seen in Figure 4.11. It is represented by a circle containing the user's profile picture.

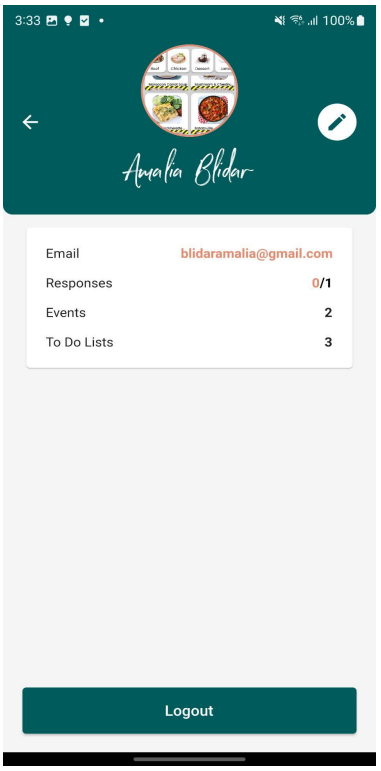



Figure 4.12. Profile Screen

Clicking on the picture will redirect the user to the Profile Screen, which displays important information such as the user's name, email, and details about their events. The Profile Screen also shows how many invitations the user has responded to, the total number of invitations received, the number of events created, and the amount of to-do lists created, as seen in Figure 4.12.

In the top right corner of the screen, there is an edit button represented by a pencil. Clicking on this button allows the user to upload a new profile picture and change the displayed name, as presented in Figure 4.12. In edit mode, the pencil button converts into a check button, as seen in Figure 4.13, which is pressed when the user is done making changes. A confirmation popup appears to ensure that the user fully understands the action of updating their profile.

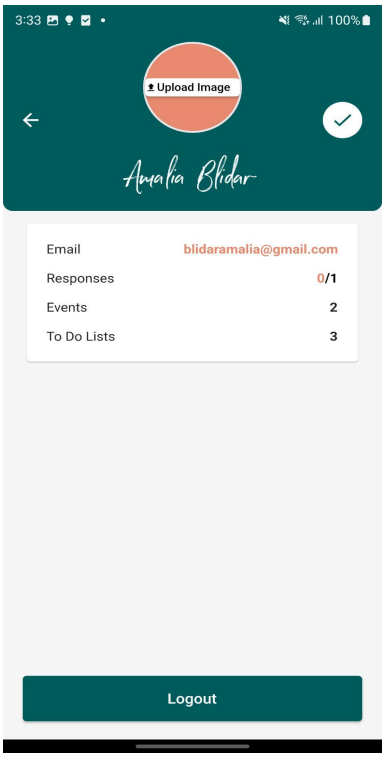
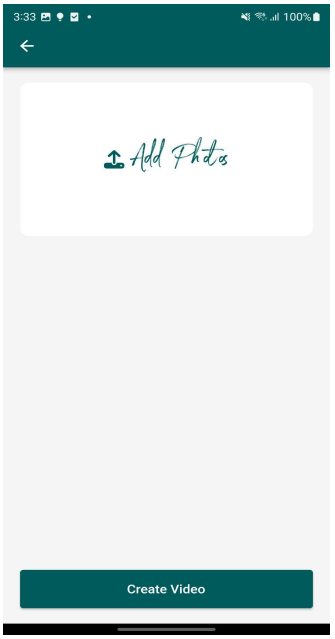



Figure 4.13. Edit Profile Screen

Pressing the "Yes" button updates the user in the database and closes the popup, revealing the updated Profile Screen.

While in edit mode, pressing the navigation back button triggers a popup that warns the user that all changes will be discarded if they accept. Pressing "Yes" will redirect the user to the profile screen with no changes made.




At the bottom of the Profile Screen, there is a logout button. Clicking on it triggers a confirmation popup, and pressing "Yes" logs the user out of the account and redirects them to the login screen.

### 4.3.6 Conversion Feature

The Conversion feature can be accessed from the center of the BottomAppBar, as it is the main feature of the application and should be easily accessible.

Figure 4.14. Conversion screen when no picture is selected

The button is represented by an intuitive icon, displaying a video camera containing the gallery icon, since the feature allows users to convert images from their gallery into videos. This button opens the gallery.

Upon clicking the button, users are taken to a screen where a large white button with an upload icon and the text "Add Photos" is displayed, as seen in Figure 4.14.

Upon selecting one or more photos from their gallery, the photos are displayed in a horizontal list of squares, allowing users to view all the photos selected, as presented in Figure 4.15.

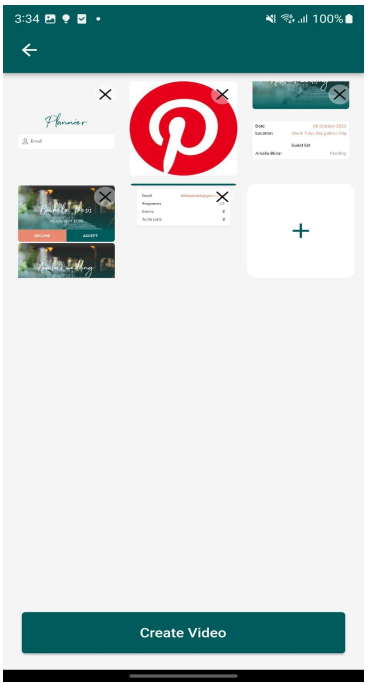



Figure 4.15. Conversion screen with selected photos

Each photo has an 'x' icon located at the top right corner, which allows the user to delete the image from the video that they are about to create. At the end of the list, a plus button is displayed that enables users to add more photos by opening the gallery.

Just dragging and dropping the images into the desired order will change their order.

The "Make Video" button is located at the bottom of the screen, and after the photos have been inserted and placed, the user can click it to start the creation process. A spinning circle will be shown on the button until the operation is finished.

The finished product will be kept locally and made accessible for viewing after the procedure is complete.

The "Create Video" button calls upon the custom-made FlutterMediaWriter Plugin, which has been developed specifically for this bachelor's thesis. This plugin enables the user to access the native methods of converting images to videos. Further information regarding the theoretical background and technical implementation of the plugin can be found in the respective chapter of this thesis.

In order to facilitate the process of adding photos and rearranging them in the Conversion feature, a third-party library was employed. Specifically, the multi_image_picker_view library was utilized to provide the necessary functionality. By incorporating this library into the Conversion feature, the application is able to provide a user-friendly experience for creating videos from multiple images.

### 4.3.7 Home Feature

Users can access information about their next forthcoming event, including a countdown displaying the number of days until the event, simply on the application's home page, shown in Figure 4.16. Users can keep track of how long they have left to make preparations thanks to this. There is a widget that displays the first invitation recorded in the database or any waiting invitations, if any, in addition to the countdown. Directly from the home page, users may view this invitation and decide whether to accept it or reject it.

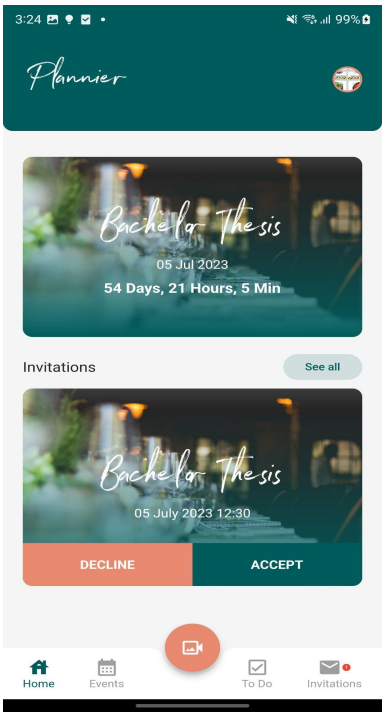



Figure 4.16. Home page

# 5. Conclusions

## 5.1. Future development possibilities

When considering the possibilities for future development, there are numerous areas that can be explored to improve the application's functionality and user experience.

In terms of the plugin component, one potential improvement is to expand the video editing capabilities. This could involve introducing a wide range of editing features such as transitions, filters, text overlays, and special effects. Providing users with a complete set of tools to customize and enhance their videos would enable them to create professional-looking content. Additionally, incorporating audio editing capabilities, such as adjusting volume levels, adding sound effects, or synchronizing multiple audio tracks, would further enrich the editing options available to users.

Another valuable feature that could be implemented is the ability to import media from various sources. This could include integrating also with direct camera capture, allowing users to easily access and import their media files into the application for editing. Integration with external devices, such as microphones, could also be considered to provide users with more flexibility and professional-like recording capabilities.

Moving to the event management aspect of the application, there are several potential features that could be introduced. One idea is to incorporate a seating plan feature, allowing event managers to view and configure the seating arrangements within the event venue directly through the application. Additionally, introducing a shopping feature where users can explore and purchase event-related items or gifts for their upcoming events could be beneficial. Furthermore, providing informative blogs or articles on event management and organization could offer users valuable insights and tips for improving their event planning skills.

Furthermore, incorporating data analytics and reporting capabilities could provide event managers with valuable insights into attendee engagement, event performance, and feedback. This could include generating event analytics reports and gathering feedback through surveys or ratings, allowing event managers to make informed decisions and continuously improve future events.

These future development ideas aim to expand the functionality and user experience of the application in both video editing and event management aspects.

## 5.2. Summary

The successful development of the FlutterMediaWriter plugin has provided users with the ability to effortlessly convert images into videos. This plugin uses native methods to ensure optimal performance and efficiency during the conversion process.

One of the primary objectives of this Bachelor Thesis was to gain a comprehensive understanding of the YUV color space and its practical application in manual image-to-video conversion. The emphasis was placed on acquiring in-depth knowledge of the underlying principles and techniques involved in this manual conversion process.

Furthermore, the thesis aimed to explore the development of plugins for cross-platform languages, with a specific focus on Flutter. The goal was to extend the capabilities of the Flutter framework by seamlessly integrating native functionalities for image-to-video conversion. By doing so, the intention was to empower the Flutter community with access to a broader range of features and functionalities that were previously only available for native platform developers.

The integration of the image-to-video conversion within the event planning aspect of the application provided a unique and contextually relevant use case for the plugin.

Given my strong interest in the Flutter framework, the development of plugins holds significant importance as it contributes to my personal growth and enables me to actively contribute to the Flutter community. By expanding the accessibility of native functionalities within Flutter, we can speed up the development process and unlock new possibilities for developers.

# 6. References

*Audio and video codec: YUV444, YUV422, YUV420 understanding in YUV sampling format*. (2020, 03 16). Video Solution Service Provider. Retrieved 4 1, 2023, from http://m.kctd-cctv.com/knowledgeencyclopedia/24-477.html

*bloc | Dart Package*. (2018). Pub.dev. Retrieved March 30, 2023, from https://pub.dev/packages/bloc

BLoC Development Team. (2020, 11 11). *BLoC*. Bloc State Management Library. Retrieved 03 30, 2023, from https://bloclibrary.dev/#/whybloc

Dart Development Team. (2012, 3 30). *Dart*. Dart programming language. Retrieved 3 30, 2023, from https://dart.dev/overview

*Developing packages & plugins | Flutter*. (2017, 8 30). Flutter documentation. Retrieved 3 30, 2023, from https://docs.flutter.dev/development/packages-and-plugins/developing-packages

Doyle, A. (2022, July 19). *Mental Health Crisis Amongst Event Planners Is Real*. Skift Meetings. Retrieved 03 25, 2023, from https://meetings.skift.com/mental-health-cevent-planners/

The Organizeaholic. (2023, 2 1). *About – The Organizeaholic*. The Organizeaholic. Retrieved 04 1, 2023, from https://theorganizeaholic.com/about/

*YUV*. (2015, 10 10). Wikipedia. Retrieved March 29, 2023, from https://en.wikipedia.org/wiki/YUV